MTB-Fetch: Multithreading Aware Hardware Prefetching for Chip Multiprocessors

Laith M. AlBarakat, Paul V. Gratz, and Daniel A. Jiménez

Abstract—To fully exploit the scaling performance in Chip Multiprocessors, applications must be divided into semi-independent processes that can run concurrently on multiple cores within a system. One major class of such applications, shared-memory, multi-threaded applications, requires programmers insert thread synchronization primitives (i.e., locks, barriers, and condition variables) in their critical sections to synchronize data access between processes. For this class of applications, scaling performance requires balanced per-thread workloads with little time spent in critical sections. In practice, however, threads often waste significant time waiting to acquire locks/barriers in their critical sections, leading to thread imbalance and poor performance scaling. Moreover, critical sections often stall data prefetchers that mitigate the effects of long critical section stalls by ensuring data is preloaded in the core caches when the critical section is complete. In this paper we examine a pure hardware technique to enable safe data prefetching beyond synchronization points in CMPs. We show that successful prefetching beyond synchronization points requires overcoming two significant challenges in existing prefetching techniques. First, we find that typical data prefetchers are designed to trigger prefetches based on current misses. This approach this works well for traditional, continuously executing, single-threaded applications. However, when a thread stalls on a synchronization point, it typically does not produce any new memory references to trigger a prefetcher. Second, even in the event that a prefetch were to be correctly directed to read beyond a synchronization point, it will likely prefetch shared data from another core before this data has been written. While this prefetch would be considered "accurate" it is highly undesirable, because such a prefetch would lead to three extra "ping-pong" movements back and forth between private caches in the producing and consuming cores, incurring more latency and energy overhead than without prefetching. We develop a new data prefetcher, Multi-Thread B-Fetch (MTB-Fetch), built as an extension to a previous single-threaded data prefetcher. MTB-Fetch addresses both issues in prefetching for shared memory multi-threaded workloads. MTB-Fetch achieves a speedup of 9.3 percent for multi-threaded applications with little additional hardware.

Index Terms—Chip multiprocessor, hardware prefetching, multi-threading, shared memory

1 Introduction

Despite increasing transistor density, the performance and power gains that traditionally accompanied process scaling have largely ceased. This trend has manifested in the current proliferation of chip-multiprocessors (CMPs) replacing single core processors as the dominant processor design, due to their lower power consumption for similar performance, however, blithely scaling core counts with future process technologies will quickly lead to diminishing returns, particularly for shared-memory, multi-threaded applications. In these applications, core and thread-count scaling often leads to performance destroying workload imbalances [1], [2]. One of the major causes of these thread-level workload imbalances, as well as degrading performance in general, is memory latency.

Prefetching is a well-studied technique to reduce the impact of memory latency. Prior work has shown that prefetching produces substantial performance gains on typical single-threaded and multi-application workloads [3], [5], [6]. Unfortunately, multi-threaded applications typically see little to no performance benefit from existing prefetching schemes. Fig. 1 shows the speedup of multi-threaded applications under three previous prefetching schemes as well as our proposed Multi-Thread B-Fetch (MTB-Fetch) scheme. The figure shows that, at best, the performance increase of the previous schemes is marginally positive, and at worst performance is somewhat degraded despite evidence that several are memory bound [2], [7]. There are two primary reasons for the poor performance of traditional prefetching techniques on these workloads: First, most prefetchers only issue a prefetch when cache miss occurs in that core. For multi-threaded applications, where an ideal time to pre-load the cache is while a given thread is spin-lock waiting, this represents a significant wasted opportunity because spin-lock loops contain no (relevant) cache misses. Second, for those few prefetchers that issue prefetches without a triggering miss (e.g., B-Fetch [4], [5]), prefetching shared data, even with perfect accuracy, might incur excess invalidations in the event that the prefetched data is read before it is written in the producing core. This is the primary cause of B-Fetch's performance loss in the figure. No prior work we are aware of has identified and addressed these two issues in prefetching for multi-threaded applications.

We present MTB-Fetch, a data prefetching scheme designed for multi-threaded workloads. The primary contribution is in addressing the issues of prefetching beyond synchronization constructs without prematurely fetching data not yet written in the producing core. We show that the overheads of MTB-Fetch are low and have no impact on the performance of prefetching for traditional single-threaded workloads. MTB-Fetch provides a speedup of 9.3 percent for PARSEC workloads, more than doubling the speedup of the best prior technique, SMS [3].

2 Background

Shared Memory Model. With growing core counts, fully exploiting the underlying microarchitecture and achieving scaling performance of single applications requires dividing that application into independent threads that can run simultaneously across the cores within a system and take advantage of thread-level parallelism (TLP). The dominant programming model for this form of TLP is shared memory multi-threading. In this programming model, an application is broken into independent threads which share a single, coherent view of memory. Typically, these independent threads share some data to complete the task. In this model, programmers insert explicit thread synchronization primitives (i.e., locks, barriers, and condition variables) to coordinate data sharing between threads, ensuring that data produced by one thread is not read by a consuming thread before it is written and so forth.

To facilitate the construction of synchronization primitives, most architectures provide some form of read-modify-write instructions that are capable of updating (i.e., reading and writing) a memory location as an atomic operation. For example, RISC style ISAs, such as ARM and ALPHA, support Load Linked (LL) and Store Conditional (SC) instructions to implement synchronization primitives [8]. In this scheme, the LL instruction loads a block of data into the cache and marks this cache line for tracking. The following SC instruction attempts to write a new value to the same block. This write succeeds only if the block has not been referenced since the preceding LL. Any memory reference to the block from another processor between the LL and SC pair causes the SC to fail.

- *L.M. AlBarakat and P.V. Gratz are with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843-3135. E-mail: {lalbarakat, pgratz}@tamu.edu.*
- *D.A. Jiménez is with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843-3135. E-mail: djimenez@cse.tamu.edu.*

Manuscript received 26 Mar. 2018; accepted 18 May 2018. Date of publication 13 June 2018; date of current version 10 July 2018.
(Corresponding author: Laith M. AlBarakat.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2018.2847345

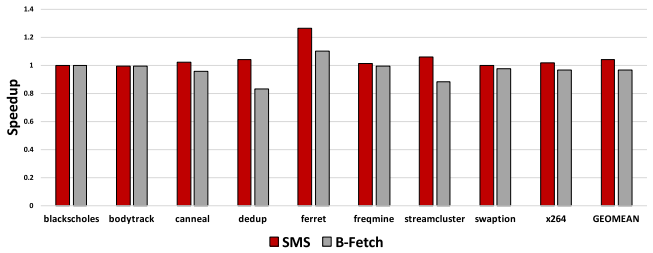

Fig. 1. Speedup of PARSEC workloads with *SMS* [3], and *B-Fetch* [4], [5], normalized against a no-prefetching baseline.

Upon failure, the locking thread will typically retry the full LL/SC pair until atomic read/modify/write success is achieved.[1]

Data Prefetching. Data prefetching is a well known technique in which the cache is pre-filled with useful data ahead of an actual demand load request coming from the processor. Typically, the prefetching opportunity is limited to waiting until a cache miss occurs, and then reading either a set of lines sequentially following the current miss [9], a set of lines following a strided pattern with respect to the current miss [10], or a set of blocks spatially around the miss [3]. More recent prefetchers attempt to predict complex, irregular access patterns [3], [6], [11], [12], [13], [14]. While these methods show significant benefit, they are inherently reactive, waiting until a cache miss occurs before they initiate prefetches down the speculated path.

Some prefetchers, such as B-Fetch [4], [5], are triggered by the fetch of a branch instruction by the processor, making them more suitable for prefetching beyond synchronization points. B-Fetch is a data cache prefetcher that employs two speculative components. It speculates on the expected path through future basic blocks, using a lookahead mechanism that relies on branch prediction to predict that execution path, and a scheme to predict the effective addresses of load instructions along that path based on the register file transformations per-basic block. B-Fetch records the variation of register contents at earlier branch instructions and uses this knowledge to predict the effective address.

Fig. 2 illustrates the overall system architecture of a system incorporating *B-Fetch* together with the additional components needed to implement MTB-Fetch. The figure shows the main CPU execution pipeline and the additional hardware for the *B-Fetch* prefetcher. *B-Fetch* forms a separate, 3-stage prefetch pipeline parallel to the main pipeline. The *B-Fetch* pipeline consists of the following stages:

1) *Branch Lookahead:* Responsible for generating the predicted path of program execution starting from the currently decoded branch.
2) *Register Lookup:* Responsible for capturing and providing information about the registers used to generate effective addresses within a given block.
3) *Prefetch Calculate:* Generates the prefetch addresses that are issued to the prefetch queue, after filtering by a per-load confidence estimator.

Multithreaded applications are just as likely to experience lost performance due to long-latency memory accesses as traditional, single threaded applications. Thus, prefetching should be a good way to improve performance. As discussed in the previous section, prefetching for multi-threaded applications produces unique challenges, in that threads waiting on synchronization typically do not induce prefetches for data beyond those synchronization points. Moreover, reckless prefetching of data beyond synchronization points could hurt performance due to premature prefetching of shared data before it has been written. *MTB-Fetch* is designed to address these issues.

1. CISC ISAs typically employ single read/modify/write atomic instructions which produce similar behavior in implementing shared memory synchronization semantics. For the purpose of discussion we focus on the LL/SC but everything discussed can apply to CISC instructions as well.

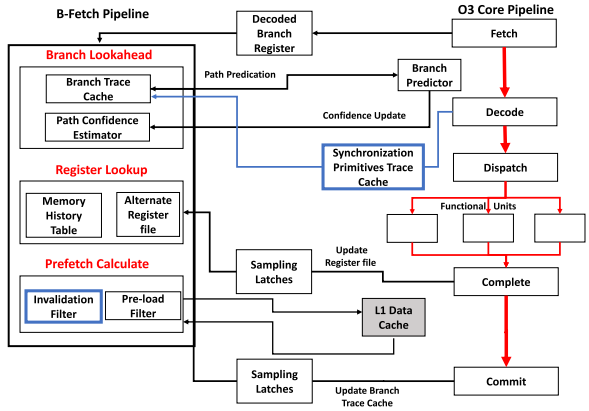

Fig. 2. *MTB-Fetch*/*B-Fetch* [4], [5] microarchitecture. *MTB-Fetch* additional components highlighted in blue.

3 Proposed Design

MTB-Fetch addresses the two issues with prefetching for multi-threaded workloads. It must continue prefetching beyond the synchronization semantic while the thread itself is busy waiting. It must also avoid issuing prefetches for shared data before it has been written.

The insight behind MTB-Fetch is to use the decode stage in the actual processor pipeline to dynamically track the synchronization primitives and identify when a thread is spinning on a lock. For a thread to acquire a lock, it must load the lock and check that no other thread is currently holding the lock. Then it must own the lock. If the thread fails to acquire the lock, it will stay in a spin loop until it successfully acquires it. Once a thread acquires the lock it is safe to execute the critical section. The thread needs to release the lock to allow other threads to execute the critical section as well. Acquiring and releasing a lock involves executing the synchronization primitive instructions LL and SC described in Section 1.

3.1 Overview

We implemented *MTB-Fetch* as an extension to the prior work *B-Fetch* prefetcher described in Section 2. To solve the first issue, prefetching beyond synchronization points, we must detect when a thread is trying to acquire and release a lock in the instruction stream, then feed the first branch instruction after releasing the lock to the B-Fetch engine to start prefetching. To this end, *MTB-Fetch* leverages the synchronization primitive instructions, LL/SC, in the dynamic instruction stream. The prefetcher identifies when a thread is spin waiting by the decoding of LL instructions. It then learns the backward branches following the LL instruction which are part of the spin once the LL/SC pair are successful and records these. Later when this synchronization point is encountered again, the prefetcher will ignore the "correct" backward branch prediction to skip ahead of the synchronization point, allowing prefetch to continue in the region beyond the critical section.

To solve the second issue, prefetch invalidation due to premature prefetching. *MTB-Fetch* keeps track of prefetches which are invalidated via the cache coherence mechanism prior to their use. This information is used to filter these "unsafe" prefetches, prohibiting them from being prefetched in the future.

3.2 System Components

Fig. 2 shows the *MTB-Fetch* microarchitecture. In particular two components, the Synchronization Primitives Trace Cache (SPTC) and Invalidation Filter are added to the original B-Fetch microarchitecture. Here we describe each.

TABLE 1
Hardware Storage Overhead in KB

Prefetcher	Component	# Entries	Size (KB)
MTB-Fetch	Branch Trace Cache	256	2.06
	Memory History Table	128	4.5
	Alternate Register File	32	0.156
	Per-Load Prefetch Filter	2048	2.25
	Additional Cache bits	-	1.37
	Prefetch Queue	32	0.156
	Path Confidence Estimator	32	0.156
	Primitives Trace Cache	256	2.06
	Invalidation Filter	2048	2.25
	TOTAL SIZE : 17.15		
B-Fetch	Branch Trace Cache	256	2.06
	Memory History Table	128	4.5
	Alternate Register File	32	0.156
	Per-Load Prefetch Filter	2048	2.25
	Additional Cache bits	-	1.37
	Prefetch Queue	32	0.156
	Path Confidence Estimator	32	0.156
	TOTAL SIZE : 12.84		
SMS	Active Generation Table	64	0.57
	Pattern History Table	16k	36
	TOTAL SIZE : 36.57		

Synchronization Primitives Trace Cache (SPTC). The SPTC dynamically captures the atomic primitives that were used to implement synchronization semantics. Each entry acts as a state machine to indicate where the beginning and the end is of critical section. Here, an LL instruction followed by a SC to the same effective address, indicates the beginning of a critical section. Once a second SC is detected, it indicates the end of a critical section. Then the first branch address after the critical section will be passed to branch lookahead in B-Fetch pipeline, so B-Fetch can predict the execution path starting from the current branch in order to prefetch data in the next basic block after the end of the synchronization semantic.

Invalidation Filter. To prevent useless prefetches wasting time, bandwidth and energy, it is crucial to reduce the number of invalidations of data prefetched but never used in the local core. The Invalidation Filter tracks data recently prefetched from another core's private caches. In the event that a cache line prefetched from another core is invalidated prior to its use by the local core, the filter notes the associated load that caused the prefetch. Future prefetches associated with that load in that basic block will be dropped before issuing under the assumption that this load is likely to lead to a premature prefetch.

3.3 Harware Cost

The additional hardware storage requirements for MTB-Fetch, B-Fetch and SMS are summarized in Table 1. Two additional components have been added to B-Fetch. In term of hardware budget Synchronization Primitives Trace Cache (SPTC) is 2.06 KB and the Invalidation Filter is 2.25 KB. To optimize the performance of SMS, we used the configuration used by Somogyi, et al. [15] and 2 KB spatial regions, a 64-entry accumulation table, and a 16K-entry pattern history table. Thus, MTB-Fetch incurs a small, 4.31 KB overhead over B-Fetch, which is still significantly less hardware state than SMS requires.

TABLE 2
Target Microarchitecture Parameters

Simulator	Gem5 Simulator, ALPHA ISA, Full System Simulation
Architecture	O3 processor, 4-wide, 192-entry ROB
ICache / DCache	32 KB, 8-way set-associative
L2Cache	256 KB, 8-way set-associative
Shared L3Cache	1024 KB per core, 16-way set-associative
Replacement Policy	LRU

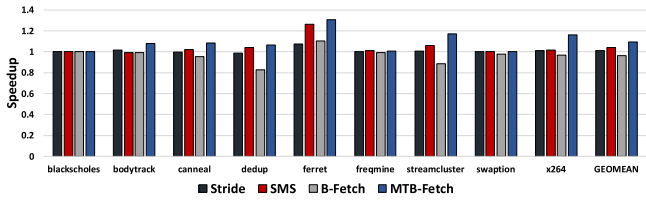

Fig. 3. Multi-threaded workload speedups.

4 Evaluation

4.1 Methodology

We used gem5 [16], a cycle accurate simulator, to evaluate MTB-Fetch. The baseline configuration is summarized in Table 2. We used a set of nine multi-threaded programs from PARSEC benchmark suite [17]. The benchmark applications represent widely used shared memory applications, which use the P-threads library to handle synchronization. The benchmark applications are cross-compiled for the ALPHA ISA and run on gem5 configured with the O3CPU CPU model (Out-of-Order) and the detailed (classic) memory model. The benchmarks were run in Full System (FS) mode.

The baseline hardware is a 4-core CMP machine with three level cache hierarchy as specified in Table 2. Each core's private cache is split into I-cache (32 KB) and D-cache(32 KB), 256 KB second level cache and 1024 KB per core third level shared cache.

MTB-Fetch results are compared against two light-weight prefetcher designs, the Stride prefetcher, SMS prefetcher, configured as described in Section 4.1 and the original B-Fetch. The Stride prefetcher was configured as in prior work [5].

4.2 Results

Fig. 3 shows the performance of each of the four prefetcher designs as the speedup compared to the baseline no-prefetching configuration. For all results, the execution time is the time spent in the region of interest (ROI). In the figure we see that MTB-Fetch provides a significant performance increase of 9.3 percent versus the baseline, more than double the performance of the closest competitor, SMS. Moreover, where the original B-Fetch showed performance regressions versus a non-prefetching baseline, MTB-Fetch improves performance for every benchmark. Interestingly, MTB-Fetch sees some of its biggest performance gains for applications where the original B-Fetch saw significant performance losses.

Fig. 4 shows the number of useful versus useless prefetches for each prefetcher. Each bar is the arithmetic average across all benchmarks. The figure illustrates several points about the behavior seen in the performance results (Fig. 3). First we see that, for the Stride prefetcher where very small performance gains are seen, generally few prefetches are issued, thus the performance gains are minimal. Interestingly for SMS, which sees some gains, there are actually fewer useful prefetches and more useless than even Stride. In this case, the useless prefetches were not enough to pollute the caches

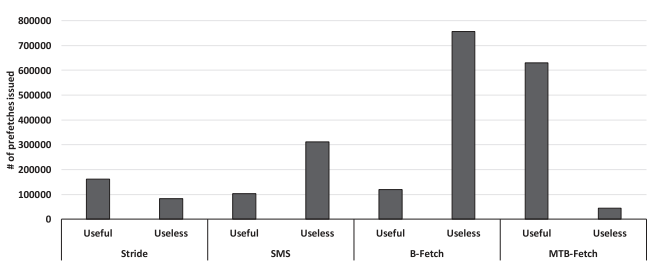

Fig. 4. Useful and useless prefetches issued, averaged across all benchmarks for each prefetcher.

significantly, while there were more useful prefetches issued on the critical thread, thus more performance gains. The original B-Fetch, while issuing slightly more useful prefetches than SMS, also issues more than twice as many useless prefetches. The original B-Fetch, often will get stuck in spin-lock loops, prefetching the lock cache line itself, which is not only useless but can cause worse performance because the lock cache line will have to be invalidated back to the core currently holding the lock. Further, for the occasions when B-Fetch does prefetch beyond the critical section (when it predicts the lock will not spin), B-Fetch often prefetches cache lines from other core's private caches before the writing core has written the data, causing performance loss as the cache line ping pong's back and forth between the private caches. In the figure, we see that MTB-Fetch, by contrast, successfully converts the majority of B-Fetch's useless prefetches into useful prefetches, this is the dominant reason why MTB-Fetch outperforms the competition on these workloads.

5 Conclusion

With increasing core-counts, shared memory multi-threading is becoming an ever more critical programming paradigm. Shared memory multi-threaded applications are similarly impacted by latency in the memory system as single-threaded applications, however, current memory prefetchers are unable to produce much performance benefit in these workloads. In this paper we identify two primary causes for poor performance in existing prefetchers for multi-threaded workloads: the inability to prefetch beyond synchronization semantics and the premature prefetching of data before it has been written in the producing core when the prefetcher is able to prefetch beyond those semantics. We then show a low overhead technique which allows prefetching beyond synchronization semantics while avoiding prefetching of data which has not yet been written by its producing thread. This scheme, MTB-Fetch, achieves a geometric mean speedup of 9.3 percent over baseline, more than twice the gains of the nearest competitor lightweight prefetcher on these workloads. As a final note, none of the additions to proposed negatively impact the single thread performance gains seen in the proposed prefetcher.

Acknowledgments

We thank the National Science Foundation, which partially supported this work through grants I/UCRC-1439722, CCF-1649242 and CCF-1216604/1332598 and Intel Corp. for their generous support.

References

[1] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Comput.*, vol. 41, no. 7, pp. 33–38, Jul. 2008.
[2] E. Fatehi and P. Gratz, "ILP and TLP in shared memory applications: A limit study," in *Proc. 23rd Int. Conf. Parallel Architectures Compilation*, 2014, pp. 113–126.
[3] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 69–80.
[4] R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for in-order processors," *IEEE Comput. Archit. Lett.*, vol. 11, no. 2, pp. 41–44, Jul. 2012.
[5] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 623–634.
[6] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2016, pp. 1–12.
[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Architectures Compilation Techn.*, Oct. 2008, pp. 72–81.
[8] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, "A new approach to exclusive data access in shared memory multiprocessors," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-97663, 1987.
[9] A. J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Comput.*, vol. 11, pp. 7–21, Dec. 1978.
[10] T. Chen and J. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, May 1995.
[11] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 247–259.
[12] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: Memory subsystem control with a unified predictor," in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 267–278.
[13] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2015, pp. 141–152.
[14] P. Michaud, "Best-offset hardware prefetching," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Mar. 2016, pp. 469–480.
[15] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proc. 33rd Annu. Int. Symp. Comput. Archit.*, 2006, pp. 252–263.
[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Princeton Univ., Princeton, NJ, Tech. Rep. TR-811–08, Jan. 2008.